Icemap

Matthew Skala

July 1, 2021

Visit the Icemap home page at http://tsukurimashou.osdn.jp/icemap.php

Contents

Introduction

This manual describes Icemap, which is a C code generator for static maps. That is a programming tool used in building computer software. It solves a specific and somewhat obscure problem. Icemap is part of the Tsukurimashou Project, which focuses on Japanese-language fonts; a need for something like it happened to come up several times in different parts of the project, so it made sense to create the tool. It is hoped that it may also be useful to others.

The use case for Icemap is as follows. Let F be a function or partial function specified as tabular data. It takes an input and produces an output, and although it might in principle be possible to calculate the output by doing arithmetic on the input, it is more convenient to just list the values. Such a function might look something like this.

in	out
3	"fizz"
5	"buzz"
6	"fizz"
9	"fizz"
10	"buzz"
12	"fizz"
15	"fizz-buzz"

Suppose it is desired to embed this function in a C program. Other code will be frequently invoking the function on different inputs and using the result in some way. We want it to be efficient, both in space and time. We don't expect the function to change often, maybe never at all; at the very least, we don't expect the user to be changing the function at run time. We will call this kind of function a *static map*.

It makes sense for the function to be hardcoded: written into C-language source code that will be processed by the compiler and made into an integral part of the program. Icemap is a tool for generating such source code.

The obvious thing would be for a human being to write the C code implementing the static map. Fine. If we are doing it only once, that may work well. When the table is small, there will be very little human work involved; possibly just cutting and pasting the table values from wherever they come from into a C file and adding some punctuation to put it into valid C syntax. We can imagine a number of different algorithms and data structures for storing and looking up the function values. In the fizz-buzz example, where inputs are integers up to 15 and outputs are strings, the simplest thing might be to build a `char *array[16]`, and do simple array indexing. If we wanted to save space we could do smarter things, like offsetting the indices, or even doing the modulo operations to *compute* instead of *looking up* the function values. This all works as long as we are working on a reasonable-sized function, willing to spend human effort on it, and we never change the function.

But consider:

- larger function tables, with perhaps a few thousand rows;
- lazy humans; and
- functions that do change from time to time, though not often.

There are a number of such tables in the Tsukurimashou Project. Most of them derive from Unicode Consortium data products. IDSgrep uses a built-in table of character widths for word-wrapping mixed Japanese-English dictionary entries in the a terminal window; this table derives from the Unicode `EastAsianWidth.txt` file. FontAnvil has a number of built-in tables for translating among character encoding formats; as well as various static maps used in parsing script files and text-based font formats.

FontAnvil's tables are in legacy code descended from an earlier package called PfaEdit. They were apparently handwritten by the original author of PfaEdit. Exactly where he got the data is not clear (certainly not well documented). Some of the code looks like it may have been generated by some kind of automated generator, but if so, that original generator has been lost. Some of the tables include clever optimizations human-designed for the specific tables and depending on the structure of the function in question. For instance, the table from Shift-

JIS to Unicode is implemented by doing some bit shifting to convert Shift-JIS to JIS, then looking up in either the JIS table or some other table depending on something, and then doing multiplication and modulo to squeeze the two-dimensional JIS array into a more memory-efficient layout. Other tables use other optimizations, such as cascading lookups through multiple layers of arrays that point at each other. The lookup processes include some built-in undocumented choices on how to handle ambiguous cases. Updating the data for a new Unicode version, or changing those choices, is effectively impossible.

In the history of PfaEdit's evolution into FontForge and then FontAnvil, there have been at least two rounds of attempts to reverse-engineer the encoding tables and create code generators to help maintenance. Those code generators sit abandoned in the FontForge source tree with fragmentary documentation. Some of FontForge's other static mapping needs (in particular, the lookup table from Unicode numbers to character names) have been spun off into separately-maintained libraries, *twice*, with an internal political dispute over which of the two spun-off libraries to actually use, and some copyright licensing implications.

Meanwhile, in IDSgrep, the character width table descends from `EastAsianWidth.txt`. The data is stored as a transition table for a finite state machine, with a fragment of C code that traverses the table for a character input until it can determine the width. There is a separate program that reads the Unicode Consortium's data file and uses binary decision diagram techniques to generate an optimized transition table. This system at least is documented, and in the event of changes to the Unicode data or differing preferences on how to resolve ambiguities, it can be maintained. However, it is highly specific to the particular function it calculates. It would require significant rewriting to work with anything else. It also makes use of certain secrets of black magic about which man was not meant to know. IDSgrep also embeds the Unicode blocks list, in a handwritten C array, with documentation but no automated support for changing it.

Icemap is intended to replace all the static maps in the Tsukurimashou project, including FontAnvil's encodings, parser tables, and character names, IDSgrep's width and Unicode block tables, and any future static maps needed by Tsukurimashou C programs, with code from a single automated generator. Automation should go all the way back to the original data sources. It should be possible to just download a new `EastAsianWidth.txt` or other original third-party data file, run `make` with an appropriate target, and have the build system update all the code without further human attention. Optimizations like the cascading lookups of PfaEdit or the minimized finite state machine of IDSgrep should be available transparently, in all maps where they are useful, without requiring human labour to implement them every time.

The function of Icemap can be seen as comparable to that of `lex`, `yacc`, or `gperf`. It is intended to run as part of the build process for some other software package, but even parties who are compiling and modifying the other software will not necessarily need to use Icemap directly; it is upstream in the overall build process and will not run during an ordinary build.

Suppose Ulrich is a third-party publisher of static map data (such as character code tables) and Alice is the original author of Fizzbuzz, a software package which needs to embed this data. Then:

- Alice writes a control file for Icemap describing how to find key-value pairs in Ulrich's distributed file format, and runs Icemap with the control file and Ulrich's data file, to generate C code implementing the static map. She puts the generated C code into the Fizzbuzz tarball and distributes it, like the rest of the Fizzbuzz C source code. The Icemap control file also goes into the tarball for use by people like Dave, below, but will not be used in ordinary compilation.
- Bob is building a copy of Fizzbuzz from source. *He does not need to use Icemap.* He just compiles the C code like any other C program.
- Carol is using a packaged binary, possibly built by Bob. She does not touch Icemap nor the C code at all.
- Dave wants to update the static map with new data from Ulrich. He needs to run Icemap with the new data file, but (assuming Ulrich's file format has not changed), he can still use Alice's control file, which tells Icemap how to read the data.
- Ellen is modifying Fizzbuzz in a more elaborate way: for instance to use static map data from Valerie instead of Ulrich, supplied in a different file format. Ellen needs to modify or

rewrite the control file and re-run Icemap.

Things notably absent from these stories: nobody needs to reverse-engineer the lookup code, write new lookup code, nor manually convert Ulrich's file format into C.

Quick start example

Icemap control language

This chapter describes the syntax and function of the Icemap control language, using a somewhat informal variant of Backus-Naur Form which I do not propose to define precisely, but which should be clear to anyone familiar with such language descriptions. The grammar described here is ambiguous (in the technical sense), but I include some informal notes on how the parser resolves the ambiguities.

To summarize the notation: ⟨*angle bracketed names*⟩ denote nonterminals (language constructions defined elsewhere); double colon equals ::= denotes the definition of a nonterminal; '`quoted typewriter font`' denotes literal single characters or C-style backslash escapes for control characters; `unquoted typewriter font` denotes strings of literal characters (keywords). The star *, plus +, and question mark ? denote respectively zero or more, one or more, and zero or one copies of the preceding symbol, as in typical Unix regular expressions. The star is not greedy. Similarly, dot . and square bracketed character classes like [`a-z`] are interpreted as in Unix regular expressions. Giving multiple definitions for a nonterminal means any of the definitions could apply; such optionality is also denoted by the pipe |.

Core syntax

Whitespace, defined as in the Unix C locale, is generally ignored in Icemap control files, except that whitespace separates tokens, and end-of-line sequences are special to the definitions of line-based comments and here documents. *Whitespace will only be mentioned in syntax rules where it is relevant.*

⟨*whitespace*⟩ ::= ⟨*ws char*⟩*

⟨*ws char*⟩ ::= '␣' | '`\t`' | '`\v`' | '`\f`' | '`\r`' | ⟨*EOL*⟩

⟨*EOL*⟩ ::= '`\n`'

Icemap accepts C-like slash-star comments and shell-like line-based comments introduced by the hash character. C comments do not have any special handling of nesting (as in C itself), and either kind of comment is treated like whitespace by the parser. Comments will not be further mentioned in syntax rules; they are implicitly allowed between tokens.

⟨*comment*⟩ ::= ⟨*C comment*⟩ | ⟨*hash comment*⟩

⟨*C comment*⟩ ::= '`/`' '`*`' .* '`*`' '`/`'

⟨*hash comment*⟩ ::= '`#`' [`^\n`]* ⟨*EOL*⟩

Statements generally end with an optional semicolon. There are some cases where using it removes some ambiguity in parsing, for instance between two successive multi-map arrow statements, but its main purpose is to enable earlier detection of typing errors. The semicolon (if not safely inside a string) causes an error anywhere except at the end of a statement. Multiple semicolons may be used.

⟨*;*⟩ ::= '`;`'*

Apart from whitespace, comments, and optional semicolons, Icemap control files are sequences of tokens. There are three kinds of tokens: "words" (which serve semantically as both keywords and identifiers); strings; and integers. Strings and integers are called value tokens; words and strings are called ids.

⟨*value*⟩ ::= ⟨*string*⟩ | ⟨*integer*⟩

⟨*id*⟩ ::= ⟨*word*⟩ | ⟨*string*⟩

Words are sequences either of alphabet letters and underscores, or of punctuation marks (basically, anything not a letter, numeral, or underscore). Punctuation-mark words are preempted by other uses of the characters in question; for instance, minus before a digit becomes part of a negative-valued integer token, and two less-thans will introduce a here document. The detailed definition of which characters count as punctuation in which contexts is thus complicated, and not given here. Note that numeric characters are not allowed in words, and will be parsed separately as integer tokens even if not whitespace-separated from a preceding word; similarly, whitespace is not necessary to separate an alphabetic word from a punctuation word. Words are not case sensitive and are *transformed to lower*

case during parsing, a fact which may be significant when defining C identifiers that will appear in Icemap's output.
⟨*word*⟩ ::= `[a-zA-Z_]+`

⟨*word*⟩ ::= `[^a-zA-Z0-9_]+`

Strings may be quoted with double quotes, in which case they accept C-like backslash escapes, or as line-based "here documents" as in the shell. I will not attempt to give BNF for exactly how escaping works inside quoted strings, but all the usual backslash escapes from C are accepted. Hexadecimal escapes like `\x1F` must have exactly two digits and octal escapes like `\037` exactly three; the more complicated end-of-sequence detection rules of C do not apply. Character codes in escapes refer to byte values (not UTF-8). Zero bytes are used as end-of-string markers internally and thus cannot usefully be included in strings.

Here documents are much as in the shell. They start with two less-thans and a tag that will be used to mark the end of the string, such as `<<EOF`. Semicolon after the tag is optional, but there must be nothing further on the line. The parser actually accepts a few other strings as tags beyond the formal definition of "word" above, but only alphabetic words are recommended for use. After defining the tag, subsequent lines of input are captured into the string, without other interpretation (in particular, comments are not removed), up to but not including a line consisting of only the tag. Note that the end-of-line characters at the ends of lines, including the last line, are included in the here document (so it is impossible to enter a string this way that does not end in an end-of-line, unless it is empty).
⟨*string*⟩ ::= ⟨*quoted string*⟩ | ⟨*here document*⟩

⟨*quoted string*⟩ ::= '"' .* '"'

⟨*line*⟩ ::= `[^\n]*` ⟨*EOL*⟩

⟨*here document*⟩ ::= '<' '<' ⟨*word*⟩ ⟨*;*⟩ ⟨*EOL*⟩
⟨*line*⟩* ⟨*word*⟩ ⟨*EOL*⟩

Integers may start with an optional minus sign, and then are hexadecimal (indicated by the prefix `0x` or `0X`), octal (indicated by an initial zero), or decimal (otherwise), much as in C.
⟨*integer*⟩ ::= ⟨*decimal int*⟩ | ⟨*octal int*⟩ | ⟨*hex int*⟩

⟨*decimal int*⟩ ::= '-'? `[1-9]` `[0-9]*`

⟨*octal int*⟩ ::= '-'? '0' `[0-7]*`

⟨*hex int*⟩ ::= '-'? '0' `[xX]` `[0-9a-fA-F]*`

At the top level, control files consist of sequences of statements.
⟨*control file*⟩ ::= ⟨*statement*⟩*

Include files

Other control files can be included with the keyword `include`. The string value given will be wildcard expanded as a shell pattern via `wordexp()`, and *all* matching files will be included. Includes may be nested.
⟨*statement*⟩ ::= `include` ⟨*string*⟩ ⟨*;*⟩

Contexts and mapping arrows

Curly braces open and close contexts. Contexts may be nested, and there is an implicit outer context surrounding the entire input. Because of the way the parser works it is technically possible (though certainly not recommended) to "close" the implicit outer context with an unbalanced closing brace; after that the parser will enter a state where the only allowed actions are to include a file and to open a new outermost context with an opening curly brace. Similarly, at the end of the input (EOF of the outermost file, outside any includes), all currently-open contexts will be automatically closed as if an appropriate number of closing braces had been inserted, but it is undesirable to depend upon this behaviour either.
⟨*statement*⟩ ::= '{' ⟨*statement*⟩* '}' ⟨*;*⟩

Contexts contain a number of different pieces of information, and in general, whenever a context is opened, it inherits copies of everything in its parent at that moment. Changes within a context affect that context, and any children or descendants opened after the change, but do not affect the parent or further ancestors.

When a context is closed, that will usually result in the creation of a map, that is, chunks of code to be written to the output C and H files. If so, Icemap increments a "leaves generated" counter on all ancestors of the context that generated the map. A non-zero value for this counter prevents a context from generating a map itself when it closes. The counter is always initialized to zero when the context opens, *not* inherited from the parent context. The effect of this behaviour is that, in the ordinary course of things, only the bottom-level contexts which have no children of their own will actually result in output to the C and H files. Higher-level contexts would typically be used to set default values to be shared among their children. Some de-

tails of how this works can be overridden by other statements described later.

The most important thing a context may contain is a collection of key-value pairs, where the key and the value may each be an integer or a string. The arrow statement adds a single such pair to the current context.
⟨*statement*⟩ ::= ⟨*value*⟩ '`-`' '`>`' ⟨*value*⟩ ⟨*;*⟩

It is also possible to add several pairs at once, by listing all the keys followed by `=>` and the corresponding values. The two lists may each contain single strings or integers, and ranges of integers indicated by a start and end with `..` in between, which is equivalent to listing all the integers from the start to the end inclusive as single items. The list of keys and list of values must contain the same number of items after evaluation of ranges.
⟨*statement*⟩ ::= ⟨*vrange*⟩* '`=`' '`>`' ⟨*vrange*⟩* ⟨*;*⟩

⟨*vrange*⟩ ::= ⟨*value*⟩

⟨*vrange*⟩ ::= ⟨*integer*⟩ '`.`' '`.`' ⟨*integer*⟩

Generating maps

The `id` statement sets a C-language identifier for the generated mapping code, to allow distinguishing multiple maps in the same source files. The default is "`map`." The value may be specified as an unquoted word token, in which case it must be valid as such, or as a string, in which case as far as Icemap is concerned it may be anything. To be useful, it ought to consist of characters accepted in C identifiers.
⟨*statement*⟩ ::= `id` ⟨*id*⟩ ⟨*;*⟩

The `generate` statement chooses the algorithm used by the mapping code. The "`nothing`" option means no map will be generated by the current context, and the generated-leaves counters of ancestor contexts will not be incremented; that may be useful in a context created for its side effects and not intended to be used directly as a map.

FIXME talk about available choices

The special contexts created inside `remap` statements do not generate maps directly nor in the ordinary way. The remapping behaviour is implemented by setting the same internal field that would be set by `generate` to a special value not available through the control-file `generate` statement. The `generate` statement should not be used inside such contexts; if used, it will convert the inner context into an ordinary child context as if "`remap`" had not been in force.

If no `generate` value is chosen for an ordinary context that would normally generate a map, then Icemap will attempt to choose one automatically that is appropriate to the types and ranges of keys and values in the map.

Using `generate` in a context resets that context's leaves generated counter to zero, so that it *will* in fact generate a map, even if it had children, provided nothing else changes. This behaviour creates two important use cases for `generate` in a parent context: use it before defining the children, and it will set a default map type for them while the parent generates no map of its own; or use `generate` after the children to force the parent to generate a map of its own and set the type of that map.
⟨*statement*⟩ ::= `generate` ⟨*maptype*⟩ ⟨*;*⟩

⟨*maptype*⟩ ::= `basic_array` | `cascade` | `wide_cascade` | `nothing`

Writing to files

Icemap generates its map code into two files, a C file (C language implementation) and an H file (C language header). The names for these files can be set by the `cfile` and `hfile` statements in the control file (whose values are automatically inherited into child contexts). Global default filenames can also be set with command-line options when Icemap is invoked, but the statements within the control file override the command-line options. It is a fatal error to do something that requires writing to the C and H files if names for them have not been set in at least one of these places.
⟨*statement*⟩ ::= `cfile` ⟨*string*⟩ ⟨*;*⟩

⟨*statement*⟩ ::= `hfile` ⟨*string*⟩ ⟨*;*⟩

Arbitrary chunks of C source code, such as licensing comments and header-file includes, can be included in the C and H files with the `cwrite` and `hwrite` statements. Note that these take effect immediately, using the current values of the C and H filenames, when the write statements are parsed—even if the current context is a `generate nothing` context or similar, and creates no map when it closes.
⟨*statement*⟩ ::= `cwrite` ⟨*string*⟩ ⟨*;*⟩

⟨*statement*⟩ ::= `hwrite` ⟨*string*⟩ ⟨*;*⟩

Built-in mappings

Icemap offers built-in encoding and decoding of UTF-8. These statements are intended for use inside `remap` blocks, to convert a mapping table in the parent context from referring to Unicode code point

numbers (integers) to the literal characters (short strings) or vice versa. The `encode utf-8` statement is equivalent to a sequence of 1114112 arrow statements mapping each single-character (possibly multi-byte) string as a key to its equivalent code point number (as an integer), and the `decode utf8` statement is equivalent to the same sequence of arrows in the opposite direction. Note that "`utf-8`" is a bit of syntactic trickery; the parser actually looks for the word token `utf` followed by an integer token that could be equal to 8 or -8. Omitting the space between them, and spelling the eight in decimal instead of some other way, are just suggestions for making the code readable.

⟨*statement*⟩ ::= `encode utf` ⟨*integer*⟩ ⟨*;*⟩

⟨*statement*⟩ ::= `decode utf` ⟨*integer*⟩ ⟨*;*⟩

⟨*statement*⟩ ::= `keytype` ⟨*id*⟩ ⟨*;*⟩

⟨*statement*⟩ ::= `valtype` ⟨*id*⟩ ⟨*;*⟩

⟨*statement*⟩ ::= `parserx` ⟨*id*⟩ ⟨*;*⟩

⟨*statement*⟩ ::= `skiprx` ⟨*id*⟩ ⟨*;*⟩

⟨*statement*⟩ ::= `rxparse` ⟨*FIXME*⟩ ⟨*;*⟩

⟨*statement*⟩ ::= `priority` ⟨*priority*⟩ ⟨*;*⟩

⟨*priority*⟩ ::= `error` | `first` | `last` | `min` | `max`

⟨*statement*⟩ ::= `quote` ⟨*FIXME*⟩ ⟨*;*⟩

⟨*statement*⟩ ::= `remap` ⟨*rmdir*⟩ '{' ⟨*statement*⟩* '}' ⟨*;*⟩

⟨*rmdir*⟩ ::= `values` | `vals` | `keys`

Index